Approaches to (Instruction-Level) Concurrency

- Pipelining
- Fine-Grained Multithreading
- Out-of-order Execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Systolic Arrays
- Decoupled Access Execute
- SIMD Processing (Vector and array processors, GPUs)
VLIW Architectures
(Very Long Instruction Word)
VLIW Concept

- Superscalar
  - **Hardware** fetches multiple instructions and checks dependencies between them

- VLIW (Very Long Instruction Word)
  - **Software (compiler) packs independent instructions** in a larger “instruction bundle” to be fetched and executed concurrently
  - Hardware fetches and executes the instructions in the bundle concurrently

- **No need for hardware dependency checking** between concurrently-fetched instructions in the VLIW model

- ELI: Enormously longword instructions (512 bits)
VLIW (Very Long Instruction Word)

- A very long instruction word consists of multiple independent instructions packed together by the compiler.
  - Packed instructions can be logically unrelated (contrast with SIMD/vector processors, which we will see soon).

- Idea: Compiler finds independent instructions and statically schedules (i.e. packs/bundles) them into a single VLIW instruction.

- Traditional VLIW Characteristics
  - Multiple instruction fetch/execute, multiple functional units
  - All instructions in a bundle are executed in lock step
  - Instructions in a bundle statically aligned to be directly fed into the functional units
VLIW Performance Example (2-wide bundles)

$lw$  $t0, 40($s0)  $add$  $t1, $s1, $s2
$sub$  $t2, $s1, $s3  $and$  $t3, $s3, $s4
$or$  $t4, $s1, $s5  $sw$  $s5, 80($s0)

Ideal IPC = 2

Actual IPC = 2 (6 instructions issued in 3 cycles)
VLIW Lock-Step Execution

- **Lock-step (all or none) execution**
  - If any operation in a VLIW instruction stalls, all concurrent operations stall

- In a **truly VLIW machine**:
  - the compiler handles all dependency-related stalls
  - hardware does **not** perform dependency checking
  - What about variable latency operations? Memory stalls?
Abstract

Multiprocessors and vector machines, the only successful parallel architectures, have coarse-grained parallelism that is hard for compilers to take advantage of. We've developed a new fine-grained parallel architecture and a compiler that together offer order-of-magnitude speedups for ordinary scientific code.

In the future, and we're building a VLIW machine, the ELI (Enormously Long Instructions) to prove it.

In this paper we'll describe some of the compilation techniques used by the Bulldog compiler. The ELI project and the details of Bulldog are described elsewhere [4, 6, 7, 15, 17].
VLIW Philosophy & Principles

- Philosophy similar to RISC (simple instructions and hardware)
  - Except multiple instructions in parallel

- RISC (John Cocke+, 1970s, IBM 801 minicomputer)
  - Compiler does the hard work to translate high-level language code to simple instructions (John Cocke: control signals)
    - And, to reorder simple instructions for high performance
  - Hardware does little translation/decoding $\rightarrow$ very simple

- VLIW (Josh Fisher, ISCA 1983)
  - Compiler does the hard work to find instruction level parallelism
  - Hardware stays as simple and streamlined as possible
    - Executes each instruction in a bundle in lock step
    - Simple $\rightarrow$ higher frequency, easier to design
VLIW Philosophy and Properties

More formally, VLIW architectures have the following properties:

There is one central control unit issuing a single long instruction per cycle.

Each long instruction consists of many tightly coupled independent operations.

Each operation requires a small, statically predictable number of cycles to execute.

Operations can be pipelined. These properties distinguish VLIWs from multiprocessors (with large asynchronous tasks) and dataflow machines (without a single flow of control, and without the tight coupling). VLIWs have none of the required regularity of a vector processor, or true array processor.

Commercial VLIW Machines

- **Multiflow TRACE**, Josh Fisher (7-wide, 28-wide)
- **Cydrome Cydra 5**, Bob Rau
- **Transmeta Crusoe**: x86 binary-translated into internal VLIW
- **TI C6000, Trimedia, STMicro (DSP & embedded processors)** and some ATI/AMD GPUs
  - Most successful commercially

- **Intel IA-64**
  - Not fully VLIW, but based on VLIW principles
  - EPIC (Explicitly Parallel Instruction Computing)
  - Instruction bundles can have dependent instructions
  - A few bits in the instruction format specify explicitly which instructions in the bundle are dependent on which other ones
VLIW Tradeoffs

- **Advantages**
  - + No need for dynamic scheduling hardware → **simple hardware**
  - + No need for dependency checking within a VLIW instruction → **simple hardware** for multiple instruction issue + no renaming
  - + No need for instruction alignment/distribution after fetch to different functional units → **simple hardware**

- **Disadvantages**
  - -- **Compiler** needs to find N independent operations per cycle
    - -- If it cannot, inserts **NOPs** in a VLIW instruction
    - -- Parallelism loss AND code size increase
  - -- **Recompilation required** when execution width (N), instruction latencies, functional units change (Unlike superscalar processing)
  - -- **Lockstep execution** causes independent operations to stall
    - -- No instruction can progress until the longest-latency instruction completes
VLIW Summary

- VLIW simplifies hardware, but requires complex compiler techniques
- Solely-compiler approach of VLIW has several downsides that reduce performance
  -- Too many NOPs (not enough parallelism discovered)
  -- Static schedule intimately tied to microarchitecture
    -- Code optimized for one generation performs poorly for next
  -- No tolerance for variable or long-latency operations (lock step)

++ Most compiler optimizations developed for VLIW employed in optimizing compilers (for superscalar compilation)
  - Enable code optimizations
++ VLIW very successful when parallelism is easier to find by the compiler (traditionally embedded markets, DSPs, GPUs)
Example Work: Trace Scheduling

Trace Scheduling Loop-Free Code

(a) A flow graph, with each block representing a basic block of code. (b) A trace picked from the flow graph. (c) The trace has been scheduled but it hasn't been relinked to the rest of the code. (d) The sections of unscheduled code that allow relinking.

ABSTRACT

By compiling ordinary scientific applications programs with a radical technique called trace scheduling, we are generating code for a parallel machine that will run these programs faster than an equivalent sequential machine — we expect 10 to 30 times faster.

Trace scheduling generates code for machines called Very Long Instruction Word architectures. In Very Long Instruction Word machines, many statically scheduled, tightly coupled, fine-grained operations execute in parallel within a single instruction stream. VLIWs are more parallel extensions of several current architectures.

These current architectures have never cracked a fundamental barrier. The speedup they get from parallelism is never more than a factor of 2 to 3. Not that we couldn't build more parallel machines of this type; but until trace scheduling we didn't know how to generate code for them. Trace scheduling finds sufficient parallelism in ordinary code to justify thinking about a highly parallel VLIW.

At Yale we are actually building one. Our machine, the ELI-512, has a horizontal instruction word of over 500 bits and will be able to execute RISC-level operation 80 times faster.

are presented in this paper. How do we put enough tests in each cycle without making the machine too big? How do we put enough memory references in each cycle without making the machine too slow?

WHAT IS A VLIW?

Everyone wants to use cheap hardware in parallel to speed up computation. One obvious approach would be to take your favorite Reduced Instruction Set Computer, let it be capable of executing 10 to 30 RISC-level operations per cycle controlled by a very long instruction word. (In fact, call it a VLIW.) A VLIW looks like very parallel horizontal microcode.

More formally, VLIW architectures have the following properties:

There is one central control unit issuing a single long instruction per cycle.

Each long instruction consists of many tightly coupled independent operations.

Each operation requires a small, statically predictable number of cycles to execute.

Operations can be pipelined. These properties distinguish
The Bulldog VLIW Compiler

Chapter 1: My Thesis

Figure 1.5. The Bulldog compiler.
Another Example Work: Superblock

The Superblock: An Effective Technique for VLIW and Superscalar Compilation

Wen-mei W. Hwu    Scott A. Mahlke    William Y. Chen    Pohua P. Chang
Nancy J. Warter    Roger A. Bringmann    Roland G. Ouellette    Richard E. Hank
Tokuzo Kiyohara    Grant E. Haab    John G. Holm    Daniel M. Lavery *


Lecture Video on Static Instruction Scheduling

- [https://www.youtube.com/watch?v=isBEVkJgGA](https://www.youtube.com/watch?v=isBEVkJgGA)
IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors

Pohua P. Chang  Scott A. Mahlke  William Y. Chen  Nancy J. Warter  Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois
Urbana, IL 61801

The performance of multiple-instruction-issue processors can be severely limited by the compiler’s ability to generate efficient code for concurrent hardware. In the IMPACT project, we have developed IMPACT-I, a highly optimizing C compiler to exploit instruction level concurrency. The optimization capabilities of the IMPACT-I C compiler are summarized in this paper. Using the IMPACT-I C compiler, we ran experiments to analyze the performance of multiple-instruction-issue processors executing some important non-numerical programs. The multiple-instruction-issue processors achieve solid speedup over high-performance single-instruction-issue processors.
Another Example Work: Hyperblock

Effective Compiler Support for Predicated Execution Using the Hyperblock

Scott A. Mahlke  David C. Lin*  William Y. Chen  Richard E. Hank  Roger A. Bringmann

Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801

Lecture Video on Static Instruction Scheduling

- https://www.youtube.com/watch?v=isBEVkJgGA

Lecture on Static Instruction Scheduling

https://www.youtube.com/onurmutlulectures
Lectures on Static Instruction Scheduling

- Computer Architecture, Spring 2015, Lecture 16
  - Static Instruction Scheduling (CMU, Spring 2015)
  - [https://www.youtube.com/watch?v=isBEVkJgGA&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=18](https://www.youtube.com/watch?v=isBEVkJgGA&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=18)

- Computer Architecture, Spring 2013, Lecture 21
  - Static Instruction Scheduling (CMU, Spring 2013)
  - [https://www.youtube.com/watch?v=XdDUn2WtkRg&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=21](https://www.youtube.com/watch?v=XdDUn2WtkRg&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=21)
A More Compact Version…

Superblock Code Optimization Example

Original Code

Code After Superblock Formation

Code After Common Subexpression Elimination

18-740 Computer Architecture - Advanced Branch Prediction - Lecture 5
4,696 views - Sep 23, 2015

Carnegie Mellon Computer Architecture
25K subscribers

Lecture 5: Advanced Branch Prediction
Lecturer: Prof. Onur Mutlu (http://users.ece.cmu.edu/~omutlu/)
Date: September 16, 2014.

Lecture 5 slides (pdf): http://www.ece.cmu.edu/~ece740/f15/ii...
Lecture 5 slides (ppt): http://www.ece.cmu.edu/~ece740/f15/ii...

https://www.youtube.com/onurmutlulectures
A More Compact Version…

- Computer Architecture, Spring 2015, Lecture 5
  - Advanced Branch Prediction (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=yDjsr-jTOtk&list=PL5PHm2jkkXmgVhh8CHAu9N76TShJqfYDt&index=4
Aside: ISA Translation

- One can translate from one ISA to another *internal-ISA* to get to a better tradeoff space
  - Programmer-visible ISA (virtual ISA) $\rightarrow$ Implementation ISA
  - Complex instructions (CISC) $\rightarrow$ Simple instructions (RISC)
  - Scalar ISA $\rightarrow$ VLIW ISA

- Examples
  - Intel’s and AMD’s x86 implementations translate x86 instructions into programmer-invisible microoperations (simple instructions) in hardware
  - Transmeta’s x86 implementations translated x86 instructions into “secret” VLIW instructions in software (code morphing software)

- Think about the tradeoffs
Transmeta: x86 to VLIW Translation

Figure 5. The Code Morphing software mediates between x86 software and the Crusoe processor.


https://www.wikiwand.com/en/Transmeta_Efficeon
https://classes.engineering.wustl.edu/cse362/images/c/c7/Paper_aklaiber_19jan00.pdf
Recall: Semantic Gap

- How close instructions & data types & addressing modes are to high-level language (HLL)

- Easier mapping of HLL to ISA
  - Less work for software designer
  - More work for hardware designer
  - Optimization burden on HW

- Harder mapping of HLL to ISA
  - More work for software designer
  - Less work for hardware designer
  - Optimization burden on SW
Recall: How to Change the Semantic Gap Tradeoffs

- Translate from one ISA into a different “implementation” ISA

SW, translator, and HW can all perform operation re-ordering
Transmeta: x86 to VLIW Translation

Figure 5. The Code Morphing software mediates between x86 software and the Crusoe processor.


https://www.wikiwand.com/en/Transmeta_Efficeon
https://classes.engineering.wustl.edu/cse362/images/c/c7/Paper_aklaiber_19jan00.pdf
Another Example: Rosetta 2 Binary Translator

Rosetta 2  [ edit ]

In 2020, Apple announced Rosetta 2 would be bundled with macOS Big Sur, to aid in the Mac transition to Apple silicon. The software permits many applications compiled exclusively for execution on x86-64-based processors to be translated for execution on Apple silicon.[2][8]

In addition to the just-in-time (JIT) translation support, Rosetta 2 offers ahead-of-time compilation (AOT), with the x86-64 code fully translated, just once, when an application without a universal binary is installed on an Apple silicon Mac.[9]

Rosetta 2's performance has been praised greatly.[10][11] In some benchmarks, x86-64-only programs performed better under Rosetta 2 on a Mac with an Apple M1 SOC than natively on a Mac with an Intel x86-64 processor. One of the key reasons why Rosetta 2 provides such high level of translation efficiency is the support of x86-64 memory ordering in Apple M1 SOC.[12]

Although Rosetta 2 works for most software, some software doesn't work at all[13] or is reported to be "sluggish".[14] A lot of software can be made compatible with the new Macs by the vendor recompiling the software, often a simple task; while for some software (such as software that includes assembly language code, or that generates machine code), the changes to make them work aren't simple and cannot be automated.

Similar to the first version, Rosetta 2 does not normally require user intervention. When a user attempts to launch an x86-64-only application for the first time, macOS prompts them to install Rosetta 2 if it is not already available. Subsequent launches of x86-64 programs will execute via translation automatically. An option also exists to force a universal binary to run as x86-64 code through Rosetta 2, even on an ARM-based machine.[15]
Apple M1, 2021

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested
The Secret of Denver: Binary Translation & Code Optimization

As we alluded to earlier, NVIDIA’s decision to forgo a traditional out-of-order design for Denver means that much of Denver’s potential is contained in its software rather than its hardware. The underlying chip itself, though by no means simple, is at its core a very large in-order processor. So it falls to the software stack to make Denver sing.

Accomplishing this task is NVIDIA’s dynamic code optimizer (DCO). The purpose of the DCO is to accomplish two tasks: to translate ARM code to Denver’s native format, and to optimize this code to make it run better on Denver. With no out-of-order hardware on Denver, it is the DCO’s task to find instruction level parallelism within a thread to fill Denver’s many execution units, and to reorder instructions around potential stalls, something that is no simple task.

The DCO system employed in the Denver CPU is codesigned software that extends ideas from prior system-level binary translators. The primary function is to execute the user’s code. The secondary function is to profile execution, create, optimize, and manage regions of tens to thousands of ARM instructions to form equivalent microcode-optimized regions that execute efficiently on the underlying microarchitecture.
DEVER: Nvidia’s First 64-bit ARM Processor

Nvidia’s first 64-bit ARM processor, code-named Denver, leverages a host of new technologies, such as dynamic code optimization, to enable high-performance mobile computing. Implemented in a 28-nm process, the Denver CPU can attain clock speeds of up to 2.5 GHz. This article outlines the Denver architecture, describes its technological innovations, and provides relevant comparisons against competing mobile processors.

Codesigning a hardware processor with a DCO software system creates both additional validation exposure and benefits. The DCO system can be upgraded in the field to address functionality, performance, or security issues.

The Denver hardware decoder provides a mechanism for periodically profiling recently taken branches. This branch history is moved into a shared buffer that can be processed from other cores, thereby minimizing the latency of the interruption. The DCO system will then run a thread that uses this profile to evaluate the dynamic properties of code executing and to assemble a picture of which code regions are hottest across all the processors. On finding sufficiently hot code, the DCO system will begin an optimization process to turn this input ARM code into a microcode execution region. The optimization process uses well-known traditional and more speculative compiler techniques to reduce work and increase efficiency of execution on the underlying skewed pipeline. To keep the latency of interruptions to a minimum, the optimizer thread is time-sliced with ARM execution (if any) and runs in a mode that can be quickly interrupted.
There Is A Lot More to Cover on ISAs
There Is A Lot More to Cover on ISAs

There are several aspects of ISAs (Instruction Set Architectures) that were not covered in the lecture. One of the key aspects is the number of registers in an ISA.

- **Affects:**
  - Number of bits used for encoding register address
  - Number of values kept in fast storage (register file)
  - (uarch) Size, access time, power consumption of register file

- **Large number of registers:**
  - Enables better register allocation (and optimizations) by compiler → fewer saves/restores
  - Larger instruction size
  - Larger register file size

For more details, you can watch the associated lecture on YouTube at [https://www.youtube.com/onurmutlulectures](https://www.youtube.com/onurmutlulectures).


28,806 views • Jan 23, 2015

Carnegie Mellon Computer Architecture
22.8k subscribers

Lecture 4. ISA Tradeoffs (cont.) & MIPS ISA
Lecturer: Kevin Chang (http://users.ece.cmu.edu/~kevincha/)
Date: Jan 21th, 2015
Detailed Lectures on ISAs & ISA Tradeoffs

- Computer Architecture, Spring 2015, Lecture 3
  - ISA Tradeoffs (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=QKdiZSfwg-g&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=3

- Computer Architecture, Spring 2015, Lecture 4
  - ISA Tradeoffs & MIPS ISA (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=RBgeCCW5Hjs&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=4

- Computer Architecture, Spring 2015, Lecture 2
  - Fundamental Concepts and ISA (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=NpC39uS4K4o&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=2

https://www.youtube.com/onurmutululectures
Approaches to (Instruction-Level) Concurrency

- Pipelining
- Fine-Grained Multithreading
- Out-of-order Execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Systolic Arrays
- Decoupled Access Execute
- SIMD Processing (Vector and array processors, GPUs)
Readings for Today

- **Required**

- **Recommended**
Readings for Next Week

- **Required**

- **Recommended**
Systolic Arrays
Systolic Arrays: Motivation

- **Goal:** design an accelerator that has
  - Simple, regular design (keep # unique parts small and regular)
  - High concurrency → high performance
  - Balanced computation and I/O (memory) bandwidth

- **Idea:** Replace a single processing element (PE) with a regular array of PEs and carefully orchestrate flow of data between the PEs
  - such that they collectively transform a piece of input data before outputting it to memory

- **Benefit:** Maximizes computation done on a single piece of data element brought from memory
Systolic Arrays

Memory: heart
Data: blood
PEs: cells
Memory pulses data through PEs

Figure 1. Basic principle of a systolic system.

Why Systolic Architectures?

- **Idea:** Data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory.

- Similar to blood flow: heart $\rightarrow$ many cells $\rightarrow$ heart
  - Different cells “process” the blood
  - Many veins operate simultaneously
  - Can be many-dimensional

- **Why?** Special purpose accelerators/architectures need
  - Simple, regular design (keep # unique parts small and regular)
  - High concurrency $\rightarrow$ high performance
  - Balanced computation and I/O (memory) bandwidth
Systolic Architectures

- Basic principle: Replace a single PE with a regular array of PEs and carefully orchestrate flow of data between the PEs
  - Balance computation and memory bandwidth

- Differences from pipelining:
  - These are individual PEs
  - Array structure can be non-linear and multi-dimensional
  - PE connections can be multidirectional (and different speed)
  - PEs can have local memory and execute kernels (rather than a piece of the instruction)
Systolic Computation Example

- **Convolution**
  - Used in filtering, pattern matching, correlation, polynomial evaluation, etc ...
  - Many image processing tasks
  - Machine learning: up to hundreds of convolutional layers in Convolutional Neural Networks (CNN)

---

Given the sequence of weights \( \{ w_1, w_2, \ldots, w_k \} \) and the input sequence \( \{ x_1, x_2, \ldots, x_n \} \), compute the result sequence \( \{ y_1, y_2, \ldots, y_{n+1-k} \} \) defined by

\[
y_i = w_1 x_i + w_2 x_{i+1} + \ldots + w_k x_{i+k-1}
\]
LeNet-5, a Convolutional Neural Network for Hand-Written Digit Recognition

This is a 1024*8 bit input, which will have a truth table of $2^{8196}$ entries.
An Example of 2D Convolution

Structure information
- Input: 5*5 (blue)
- Kernel (filter): 3*3 (grey)
- Output: 5*5 (green)

Computation information
- Stride: 1
- Padding: 1 (white)

Output Dim = (Input + 2*Padding - Kernel) / Stride + 1
An Example of 2D Convolution

Input Layer

CNN kernel

Output Layer

![Diagram showing an example of 2D convolution with an input layer, a CNN kernel, and an output layer.](image-url)
Convolutional Neural Networks: Demo

LeNet-5, convolutional neural networks

Convolutional Neural Networks are a special kind of multi-layer neural networks. Like almost every other neural networks they are trained with a version of the back-propagation algorithm. Where they differ is in the architecture.

Convolutional Neural Networks are designed to recognize visual patterns directly from pixel images with minimal preprocessing. They can recognize patterns with extreme variability (such as handwritten characters), and with robustness to distortions and simple geometric transformations.

LeNet-5 is our latest convolutional network designed for handwritten and machine-printed character recognition. Here is an example of LeNet-5 in action.

Many more examples are available in the column on the left:

Several papers on LeNet and convolutional networks are available on my publication page:

[LeCun et al., 1998]
Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner.

[Bottou et al., 1997]
L. Bottou, Y. LeCun, and Y. Bengio. Global training of
Implementing a Convolutional Layer with Matrix Multiplication

\[
\begin{pmatrix}
1 & 1 & 1 & 2 & 2 & 0 \\
1 & 1 & 1 & 0 & 1 & 2 \\
1 & 0 & 0 & 1 & 2 & 1 \\
1 & 2 & 1 & 2 & 1 & 0 \\
1 & 2 & 1 & 0 & 1 & 3 \\
0 & 2 & 0 & 1 & 1 & 2 \end{pmatrix} \times \begin{pmatrix}
1 & 0 & 0 & 1 & 2 & 1 & 2 & 1 & 1 & 2 & 2 & 0 \\
1 & 1 & 3 & 1 & 1 & 3 & 0 & 2 & 2 & 1 & 1 & 0 \\
0 & 2 & 0 & 1 & 1 & 2 & 0 & 1 & 1 & 2 & 2 & 0 \\
1 & 2 & 0 & 1 & 0 & 1 & 3 & 2 & 2 & 2 & 0 & 1 \\
1 & 3 & 0 & 2 & 0 & 1 & 1 & 2 & 2 & 0 & 1 & 3 \\
0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
1 & 2 & 1 & 0 & 1 & 2 & 1 & 0 & 1 & 1 & 0 & 1 \\
1 & 2 & 0 & 1 & 0 & 1 & 3 & 2 & 2 & 2 & 0 & 1 \\
2 & 1 & 3 & 1 & 1 & 3 & 0 & 2 & 2 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
1 & 3 & 0 & 2 & 0 & 1 & 1 & 2 & 2 & 0 & 1 & 3 \\
1 & 3 & 3 & 2 & 1 & 2 & 1 & 2 & 1 & 1 & 2 & 2 \end{pmatrix} = \begin{pmatrix}
12 & 18 & 10 & 20 \\
13 & 22 & 15 & 22 \end{pmatrix}
\]
In 2010, Prof. Andreas Moshovos adopted Professor Hwu’s ECE498AL Programming Massively Parallel Processors Class

Several of Prof. Geoffrey Hinton’s graduate students took the course

These students developed the GPU implementation of the Deep CNN that was trained with 1.2M images to win the ImageNet competition

Slide credit: Hwu & Kirk
Example: AlexNet (2012)

- AlexNet wins the **ImageNet classification competition** with ~10% points higher accuracy than state-of-the-art
  - Krizhevsky et al., “**ImageNet Classification with Deep Convolutional Neural Networks**”, NIPS 2012.

---

**ImageNet Classification with Deep Convolutional Neural Networks**

Alex Krizhevsky  
University of Toronto  
kriz@cs.utoronto.ca

Ilya Sutskever  
University of Toronto  
ilya@cs.utoronto.ca

Geoffrey E. Hinton  
University of Toronto  
hinton@cs.utoronto.ca

**Abstract**

We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes. On the test data, we achieved top-1 and top-5 error rates of 37.5% and 17.0% which is considerably better than the previous state-of-the-art. The neural network, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax. To make training faster, we used non-saturating neurons and a very efficient GPU implementation of the convolution operation. To reduce overfitting in the fully-connected layers we employed a recently-developed regularization method called “dropout” that proved to be very effective. We also entered a variant of this model in the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry.
Example: GoogLeNet (2014)

- Google improves accuracy by **adding more network layers**
  - From 8 in AlexNet to 22 in GoogLeNet
  - Szegedy et al., “Going Deeper with Convolutions”, CVPR 2015.

---

**Going Deeper with Convolutions**

Christian Szegedy\(^1\), Wei Liu\(^2\), Yangqing Jia\(^1\), Pierre Sermanet\(^1\), Scott Reed\(^3\), Dragomir Anguelov\(^1\), Dumitru Erhan\(^1\), Vincent Vanhoucke\(^1\), Andrew Rabinovich\(^4\)

\(^1\)Google Inc. \(^2\)University of North Carolina, Chapel Hill
\(^3\)University of Michigan, Ann Arbor \(^4\)Magic Leap Inc.

\(^1\)szegedy, jiayq, sermanet, dragomir, dumitru, vanhoucke\}@google.com
\(^2\)wliu@cs.unc.edu, \(^3\)reedscott@umich.edu, \(^4\)arabinovich@magicleap.com
**Example: ResNet (2015)**


**Deep Residual Learning for Image Recognition**

Kaiming He  
Xiangyu Zhang  
Shaoqing Ren  
Jian Sun  
Microsoft Research  
{kaiminghe, xiangyu, shaoqing, jiansun}@microsoft.com

### ImageNet experiments

<table>
<thead>
<tr>
<th>Year</th>
<th>Model</th>
<th>Layers</th>
<th>Top-5 Error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILSVRC'15</td>
<td>ResNet</td>
<td>152</td>
<td>3.57</td>
</tr>
<tr>
<td>ILSVRC'14</td>
<td>GoogleNet</td>
<td>22</td>
<td>6.7</td>
</tr>
<tr>
<td>ILSVRC'14</td>
<td>VGG</td>
<td>19</td>
<td>7.3</td>
</tr>
<tr>
<td>ILSVRC'13</td>
<td></td>
<td></td>
<td>11.7</td>
</tr>
<tr>
<td>ILSVRC'12</td>
<td>AlexNet</td>
<td>8</td>
<td>8 layers</td>
</tr>
<tr>
<td>Shallow</td>
<td></td>
<td></td>
<td>25.8</td>
</tr>
<tr>
<td>ILSVRC'11</td>
<td></td>
<td></td>
<td>28.2</td>
</tr>
</tbody>
</table>

Human: 5.1%  
First CNN
Neural Network Layer Examples

LeNet

<table>
<thead>
<tr>
<th>Image: 28 (height) × 28 (width) × 1 (channel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution with 5×5 kernel+2 padding:28×28×6</td>
</tr>
<tr>
<td>↓ sigmoid</td>
</tr>
<tr>
<td>Pool with 2×2 average kernel+2 stride:14×14×6</td>
</tr>
<tr>
<td>↓ Convolution with 5×5 kernel (no pad):10×10×16</td>
</tr>
<tr>
<td>↓ sigmoid</td>
</tr>
<tr>
<td>Pool with 2×2 average kernel+2 stride:5×5×16</td>
</tr>
<tr>
<td>↓ flatten</td>
</tr>
<tr>
<td>Dense: 120 fully connected neurons</td>
</tr>
<tr>
<td>↓ sigmoid</td>
</tr>
<tr>
<td>Dense: 84 fully connected neurons</td>
</tr>
<tr>
<td>↓ sigmoid</td>
</tr>
<tr>
<td>Dense: 10 fully connected neurons</td>
</tr>
<tr>
<td>↓ Output: 1 of 10 classes</td>
</tr>
</tbody>
</table>

AlexNet

<table>
<thead>
<tr>
<th>Image: 224 (height) × 224 (width) × 3 (channels)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution with 11×11 kernel+4 stride:54×54×96</td>
</tr>
<tr>
<td>↓ ReLu</td>
</tr>
<tr>
<td>Pool with 3×3 max. kernel+2 stride: 26×26×96</td>
</tr>
<tr>
<td>↓ Convolution with 5×5 kernel+2 pad:26×26×256</td>
</tr>
<tr>
<td>↓ ReLu</td>
</tr>
<tr>
<td>Pool with 3×3 max. kernel+2 stride:12×12×256</td>
</tr>
<tr>
<td>↓ Convolution with 3×3 kernel+1 pad:12×12×384</td>
</tr>
<tr>
<td>↓ ReLu</td>
</tr>
<tr>
<td>Convolution with 3×3 kernel+1 pad:12×12×384</td>
</tr>
<tr>
<td>↓ ReLu</td>
</tr>
<tr>
<td>Convolution with 3×3 kernel+1 pad:12×12×256</td>
</tr>
<tr>
<td>↓ ReLu</td>
</tr>
<tr>
<td>Pool with 3×3 max. kernel+2 stride:5×5×256</td>
</tr>
<tr>
<td>↓ flatten</td>
</tr>
<tr>
<td>Dense: 4096 fully connected neurons</td>
</tr>
<tr>
<td>↓ ReLu, dropout p=0.5</td>
</tr>
<tr>
<td>Dense: 4096 fully connected neurons</td>
</tr>
<tr>
<td>↓ ReLu, dropout p=0.5</td>
</tr>
<tr>
<td>Dense: 1000 fully connected neurons</td>
</tr>
<tr>
<td>↓ Output: 1 of 1000 classes</td>
</tr>
</tbody>
</table>
Systolic Computation Example: Convolution (I)

- **Convolution**
  - Used in filtering, pattern matching, correlation, polynomial evaluation, etc ...
  - Many image processing tasks
  - Machine learning: up to hundreds of convolutional layers in Convolutional Neural Networks (CNN)

Given the sequence of weights \( \{w_1, w_2, \ldots, w_k\} \)
and the input sequence \( \{x_1, x_2, \ldots, x_n\} \),
compute the result sequence \( \{y_1, y_2, \ldots, y_{n+1-k}\} \)
defined by

\[
y_i = w_1 x_i + w_2 x_{i+1} + \cdots + w_k x_{i+k-1}
\]
Systolic Computation Example: Convolution (II)

- \[ y_1 = w_1 x_1 + w_2 x_2 + w_3 x_3 \]
- \[ y_2 = w_1 x_2 + w_2 x_3 + w_3 x_4 \]
- \[ y_3 = w_1 x_3 + w_2 x_4 + w_3 x_5 \]

Figure 8. Design W1: systolic convolution array (a) and cell (b) where \( w_i \)'s stay and \( x_i \)'s and \( y_i \)'s move systolically in opposite directions.
Worthwhile to implement adder and multiplier separately to allow overlapping of add/mul executions.
Systolic Computation Example: Convolution (IV)

- One needs to **carefully orchestrate** when **data elements are input to the array**
- And when **output is buffered**

- This gets more involved when
  - Array dimensionality increases
  - PEs are less predictable in terms of latency
Example 2D Systolic Array Computation

- Multiply two 3x3 matrices (inputs)
  - Keep the final result in PE accumulators

\[
\begin{bmatrix}
c_{00} & c_{01} & c_{02} \\
c_{10} & c_{11} & c_{12} \\
c_{20} & c_{21} & c_{22} \\
\end{bmatrix}
\begin{bmatrix}
a_{00} & a_{01} & a_{02} \\
a_{10} & a_{11} & a_{12} \\
a_{20} & a_{21} & a_{22} \\
\end{bmatrix}
= 
\begin{bmatrix}
b_{00} & b_{01} & b_{02} \\
b_{10} & b_{11} & b_{12} \\
b_{20} & b_{21} & b_{22} \\
\end{bmatrix}
\]

Figure 1: A systolic array processing element

\[P = M\]
\[Q = N\]
\[R = R + M \times N\]
Two-Dimensional Systolic Arrays

Figure 11. Two-dimensional systolic arrays: (a) type R, (b) type H, and (c) type T.

To a given problem there could be both one- and two-dimensional systolic array solutions. For example, two-dimensional convolution can be performed by a one-dimensional systolic array\textsuperscript{24,25} or a two-dimensional systolic array.\textsuperscript{6} When the memory speed is more than cell speed, two-dimensional systolic arrays such as those depicted in Figure 11 should be used. At each cell cycle, all the I/O ports on the array boundaries can input or output data items to or from the memory; as a result, the available memory bandwidth can be fully utilized. Thus, the choice of a one- or two-dimensional scheme is very dependent on how cells and memories will be implemented.
Combinations

- Systolic arrays can be chained together to form powerful systems.

- This systolic array is capable of producing on-the-fly least-squares fit to all the data that has arrived up to any given moment.

Figure 12. On-the-fly least-squares solutions using one- and two-dimensional systolic arrays, with $p = 4$. 

GIVEN AN $n \times p$ MATRIX $X$ WITH $n \geq p$, AND AN $n$-VECTOR $y$, DETERMINE A $p$-VECTOR $b$ SUCH THAT $\|y - xb\|$ IS MINIMIZED.

STEP 1: ORTHOGONAL TRIANGULARIZATION
STEP 2: SOLUTION OF TRIANGULAR LINEAR SYSTEM
Systolic Arrays: Pros and Cons

- **Advantages:**
  - **Principled:** Efficiently makes use of limited memory bandwidth, balances computation to I/O bandwidth availability
  - **Specialized** (computation needs to fit PE organization/functions)
    → improved efficiency, simple design, high concurrency/performance
    → good to do more with less memory bandwidth requirement

- **Downside:**
  - **Specialized**
    → not generally applicable because computation needs to fit the PE functions/organization
More Programmability in Systolic Arrays

- Each PE in a systolic array
  - Can store multiple “weights”
  - Weights can be selected on the fly
  - Eases implementation of, e.g., adaptive filtering

- Taken further
  - Each PE can have its own data and instruction memory
  - Data memory → to store partial/temporary results, constants
  - Leads to stream processing, pipeline parallelism
    - More generally, staged execution
Figure 1. (a) The code of a loop, (b) Each iteration is split into 3 pipeline stages: A, B, and C. Iteration i comprises Ai, Bi, Ci. (c) Sequential execution of 4 iterations. (d) Parallel execution of 6 iterations using pipeline parallelism on a three-core machine. Each stage executes on one core.
Stages of Pipelined Programs

- Loop iterations are divided into code segments called **stages**
- Threads execute stages on different cores

```python
loop {
    Compute1
    Compute2
    Compute3
}
```
Pipelined File Compression Example

Figure 3. File compression algorithm executed using pipeline parallelism
Systolic Array: Advantages & Disadvantages

- **Advantages**
  - Makes *multiple uses of each data item* → reduced need for fetching/refetching → better use of memory bandwidth
  - **High concurrency**
  - Regular design (both data and control flow)

- **Disadvantages**
  - Not good at exploiting irregular parallelism
  - Relatively special purpose → need software, programmer support to be a general purpose model
Example Systolic Array: The WARP Computer

- HT Kung, CMU, 1984-1988

- Linear array of 10 cells, each cell a 10 Mflop programmable processor
- Attached to a general purpose host machine
- HLL and optimizing compiler to program the systolic array
- Used extensively to accelerate vision and robotics tasks

The WARP Computer

Figure 1: Warp system overview
The WARP Cell

Figure 2: Warp cell data path
An Example Modern Systolic Array: TPU (I)

Figure 3. TPU Printed Circuit Board. It can be inserted in the slot for an SATA disk in a server, but the card uses PCIe Gen3 x16.

Figure 4. Systolic data flow of the Matrix Multiply Unit. Software has the illusion that each 256B input is read at once, and they instantly update one location of each of 256 accumulator RAMs.

An Example Modern Systolic Array: TPU (II)

As reading a large SRAM uses much more power than arithmetic, the matrix unit uses systolic execution to save energy by reducing reads and writes of the Unified Buffer [Kun80][Ram91][Ovt15b]. Figure 4 shows that data flows in from the left, and the weights are loaded from the top. A given 256-element multiply-accumulate operation moves through the matrix as a diagonal wavefront. The weights are preloaded, and take effect with the advancing wave alongside the first data of a new block. Control and data are pipelined to give the illusion that the 256 inputs are read at once, and that they instantly update one location of each of 256 accumulators. From a correctness perspective, software is unaware of the systolic nature of the matrix unit, but for performance, it does worry about the latency of the unit.

Recall: Example 2D Systolic Array Computation

- Multiply two 3x3 matrices (inputs)
  - Keep the final result in PE accumulators

\[ \begin{bmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix} \]

Figure 1: A systolic array processing element

\[ \begin{align*}
P &= M \\
Q &= N \\
R &= R + M \times N
\]
Figure 1. TPU Block Diagram. The main computation part is the yellow Matrix Multiply unit in the upper right hand corner. Its inputs are the blue Weight FIFO and the blue Unified Buffer (UB) and its output is the blue Accumulators (Acc). The yellow Activation Unit performs the nonlinear functions on the Acc, which go to the UB.
An Example Modern Systolic Array: TPU2

4 TPU chips
vs 1 chip in TPU1

High Bandwidth Memory
vs DDR3

Floating point operations
vs FP16

45 TFLOPS per chip
vs 23 TOPS

Designed for training and inference
vs only inference
An Example Modern Systolic Array: TPU3

32GB HBM per chip vs 16GB HBM in TPU2
4 Matrix Units per chip vs 2 Matrix Units in TPU2
90 TFLOPS per chip vs 45 TFLOPS in TPU2

https://cloud.google.com/tpu/docs/system-architecture
An Example Modern Systolic Array: TPU4

New ML applications (vs. TPU3):
- Computer vision
- Natural Language Processing (NLP)
- Recommender system
- Reinforcement learning that plays Go

250 TFLOPS per chip in 2021 vs 90 TFLOPS in TPU3

1 ExaFLOPS per board

https://spectrum.ieee.org/tech-talk/computing/hardware/heres-how-googles-tpu-v4-ai-chip-stacked-up-in-training-tests
Cerebras’s Wafer Scale Engine (2019)

- The largest ML accelerator chip
- 400,000 cores

Cerebras WSE
1.2 Trillion transistors
46,225 mm²

Largest GPU
21.1 Billion transistors
815 mm²

https://www.anandtech.com/show/14758/hot-chips-31-live-blogs-cerebras-wafer-scale-deep-learning
https://www.cerebras.net/cerebras-wafer-scale-engine-why-we-need-big-chips-for-deep-learning
Cerebras’s Wafer Scale Engine-2 (2021)

- The largest ML accelerator chip
- 850,000 cores

Cerebras WSE-2
2.6 Trillion transistors
46,225 mm²

Largest GPU
54.2 Billion transistors
826 mm²

https://www.anandtech.com/show/14758/hot-chips-31-live-blogs-cerebras-wafer-scale-deep-learning
https://www.cerebras.net/cerebras-wafer-scale-engine-why-we-need-big-chips-for-deep-learning
Huge Demand for Performance & Efficiency

Exponential Growth of Neural Networks

1800x more compute
In just 2 years

Tomorrow, multi-trillion parameter models

https://www.youtube.com/watch?v=x2-qB0J7KHz
More on the Cerebras WSE

https://www.youtube.com/watch?v=x2-qB0J7KHW
Backup Slides

(for Further Study)
Issues in Fast & Wide Fetch Engines
These Issues Covered in This Lecture…

18-740 Computer Architecture - Advanced Branch Prediction - Lecture 5
4,696 views - Sep 23, 2015

Carnegie Mellon Computer Architecture
25K subscribers

Lecture 5: Advanced Branch Prediction
Lecturer: Prof. Onur Mutlu (http://users.ece.cmu.edu/~omutlu/)
Date: September 16, 2014.

Lecture 5 slides (pdf): http://www.ece.cmu.edu/~ece740/f15/l5...
Lecture 5 slides (ppt): http://www.ece.cmu.edu/~ece740/f15/l5...

https://www.youtube.com/onurmutlulectures
These Issues Covered in This Lecture…

- Computer Architecture, Spring 2015, Lecture 5
  - Advanced Branch Prediction (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=yDjsr-jTOtk&list=PL5PHm2jkkXmgVhh8CHAu9N76TShJqfYDt&index=4
Interference in Branch Predictors
An Issue: Interference in the PHTs

- Sharing the PHTs between histories/branches leads to interference
  - Different branches map to the same PHT entry and modify it
  - Interference can be positive, negative, or neutral

- Interference can be eliminated by dedicating a PHT per branch
  -- Too much hardware cost

- How else can you eliminate or reduce interference?
Reducing Interference in PHTs (I)

- Increase size of PHT

- Branch filtering
  - Predict highly-biased branches separately so that they do not consume PHT entries
  - E.g., static prediction or BTB based prediction

- Hashing/index-randomization
  - Gshare
  - Gskew

- Agree prediction
Biased Branches and Branch Filtering

- **Observation:** Many branches are biased in one direction (e.g., 99% taken)

- **Problem:** These branches *pollute* the branch prediction structures → make the prediction of other branches difficult by causing “interference” in branch prediction tables and history registers

- **Solution:** Detect such biased branches, and predict them with a simpler predictor (e.g., last time, static, ...)

Reducing Interference: Gshare

- **Idea 1:** Randomize the indexing function into the PHT such that probability of two branches mapping to the same entry reduces
  - **Gshare predictor:** GHR hashed with the Branch PC
  - Better utilization of PHT + More context information
  - Increases access latency

Reducing Interference: Agree Predictor

- Idea 2: Agree prediction
  - Each branch has a “bias” bit associated with it in BTB
    - Ideally, most likely outcome for the branch
  - High bit of the PHT counter indicates whether or not the prediction agrees with the bias bit (not whether or not prediction is taken)

+ Reduces negative interference (Why???)
-- Requires determining bias bits (compiler vs. hardware)

Why Does Agree Prediction Make Sense?

- Assume two branches have taken rates of 85% and 15%.
- Assume they conflict in the PHT

Let’s compute the probability they have opposite outcomes

- Baseline predictor:
  - P (b1 T, b2 NT) + P (b1 NT, b2 T)
  - = (85%*85%) + (15%*15%) = 74.5%

- Agree predictor:
  - Assume bias bits are set to T (b1) and NT (b2)
  - P (b1 agree, b2 disagree) + P (b1 disagree, b2 agree)
  - = (85%*15%) + (15%*85%) = 25.5%

- Works because most branches are biased (not 50% taken)
Reducing Interference: Gskew

- Idea 3: Gskew predictor
  - Multiple PHTs
  - Each indexed with a different type of hash function
  - Final prediction is a majority vote
    - Distributes interference patterns in a more randomized way (interfering patterns less likely in different PHTs at the same time)
    - More complexity (due to multiple PHTs, hash functions)


More Techniques to Reduce PHT Interference

- **The bi-mode predictor**
  - Separate PHTs for mostly-taken and mostly-not-taken branches
  - Reduces negative aliasing between them

- **The YAGS predictor**
  - Use a small tagged “cache” to predict branches that have experienced interference
  - Aims to not to mispredict them again

- **Alpha EV8 (21464) branch predictor**
Another Direction: Helper Threading

- **Idea:** Pre-compute the outcome of the branch with a separate, customized thread (i.e., a helper thread)


Issues in Wide & Fast Fetch
I-Cache Line and Way Prediction

- **Problem:** Complex branch prediction can take too long (many cycles)
- **Goal**
  - Quickly generate (a reasonably accurate) next fetch address
  - Enable the fetch engine to run at high frequencies
  - Override the quick prediction with more sophisticated prediction
- **Idea:** Get the predicted next cache line and way at the time you fetch the current cache line

- **Example Mechanism (e.g., Alpha 21264)**
  - Each cache line tells which line/way to fetch next (prediction)
  - On a fill, line/way predictor points to next sequential line
  - On branch resolution, line/way predictor is updated
  - If line/way prediction is incorrect, one cycle is wasted
Figure 3. Alpha 21264 instruction fetch. The line and way prediction (wrap-around path on the right side) provides a fast instruction fetch path that avoids common fetch stalls when the predictions are correct.
Alpha 21264 Line & Way Prediction

Issues in Wide Fetch Engines

- Wide Fetch: Fetch multiple instructions per cycle
- Superscalar
- VLIW
- SIMT (GPUs’ single-instruction multiple thread model)

Wide fetch engines suffer from the branch problem:
- How do you feed the wide pipeline with useful instructions in a single cycle?
- What if there is a taken branch in the “fetch packet”?
- What is there are “multiple (taken) branches” in the “fetch packet”?
Fetching Multiple Instructions Per Cycle

- Two problems

1. **Alignment** of instructions in I-cache
   - What if there are not enough (N) instructions in the cache line to supply the fetch width?

2. **Fetch break**: Branches present in the fetch block
   - Fetching sequential instructions in a single cycle is easy
   - What if there is a control flow instruction in the N instructions?
   - Problem: The direction of the branch is not known but we need to fetch more instructions

- These can cause effective fetch width < peak fetch width
Wide Fetch Solutions: Alignment

- **Large cache blocks**: Hope N instructions contained in the block

- **Split-line fetch**: If address falls into second half of the cache block, fetch the first half of next cache block as well
  - Enabled by banking of the cache
  - Allows sequential fetch across cache blocks in one cycle
  - Intel Pentium and AMD K5
Split Line Fetch

Cache Banking

Memory Map

Cache

Need alignment logic:
Short Distance Predicted-Taken Branches

First Iteration (Branch B taken to E)

Second Iteration (Branch B fall through to C)
Techniques to Reduce Fetch Breaks

- Compiler
  - Code reordering (basic block reordering)
  - Superblock

- Hardware
  - Trace cache

- Hardware/software cooperative
  - Block structured ISA
Basic Block Reordering

- Not-taken control flow instructions not a problem: no fetch break: make the likely path the not-taken path
- Idea: Convert taken branches to not-taken ones
  - i.e., reorder basic blocks (after profiling)
  - Basic block: code with a single entry and single exit point

- Code Layout 1 leads to the fewest fetch breaks

![Control Flow Graph]

![Code Layouts]

- Code Layout 1
- Code Layout 2
- Code Layout 3
Basic Block Reordering


**Advantages:**
- Reduced fetch breaks (assuming profile behavior matches runtime behavior of branches)
- Increased I-cache hit rate
- Reduced page faults

**Disadvantages:**
- Dependent on compile-time profiling
- Does not help if branches are not biased
- Requires recompilation
Superblock

- Idea: Combine frequently executed basic blocks such that they form a single-entry multiple exit larger block, which is likely executed as straight-line code

  + Helps wide fetch
  + Enables aggressive compiler optimizations and code reordering within the superblock

-- Increased code size
-- Profile dependent
-- Requires recompilation

Superblock Formation (I)

Is this a superblock?
**Tail duplication:**
duplication of basic blocks after a side entrance to eliminate side entrances
→ transforms a trace into a superblock.
Superblock Code Optimization Example

Original Code

\[ \text{opA: mul } r1 <- r2,3 \]
\[ \text{opB: add } r2 <- r2,1 \]
\[ \text{opC: mul } r3 <- r2,3 \]

99
1

Code After Superblock Formation

\[ \text{opA: mul } r1 <- r2,3 \]
\[ \text{opC: mov } r3 <- r1 \]
\[ \text{opB: add } r2 <- r2,1 \]
\[ \text{opC': mul } r3 <- r2,3 \]

Code After Common Subexpression Elimination

\[ \text{opA: mul } r1 <- r2,3 \]
\[ \text{opC': mul } r3 <- r2,3 \]
\[ \text{opB: add } r2 <- r2,1 \]

99
1
Techniques to Reduce Fetch Breaks

- **Compiler**
  - Code reordering (basic block reordering)
  - Superblock

- **Hardware**
  - Trace cache

- **Hardware/software cooperative**
  - Block structured ISA
Trace Cache: Basic Idea

- A trace is a sequence of executed instructions.
- It is specified by a start address and the branch outcomes of control transfer instructions.
- Traces repeat: programs have frequently executed paths.
- Trace cache idea: Store the dynamic instruction sequence in the same physical location.

(a) Instruction cache.  (b) Trace cache.
Reducing Fetch Breaks: Trace Cache

- Dynamically determine the basic blocks that are executed consecutively
- Trace: Consecutively executed basic blocks
- Idea: Store consecutively-executed basic blocks in physically-contiguous internal storage (called trace cache)

![Dynamic Instruction Stream Diagram]

- Basic trace cache operation:
  - Fetch from consecutively-stored basic blocks (predict next trace or branches)
  - Verify the executed branch directions with the stored ones
  - If mismatch, flush the remaining portion of the trace

Trace Cache: Example

- **Instruction Cache**
  - 1st BB
  - 2nd BB
  - 3rd BB
  - A

- **Trace Cache**
  - 1st BB
  - 2nd BB
  - 3rd BB
  - A
  - **hit?**

- **Line-Fill Buffer**

- **Instruction Latch**
  - 0
  - 1

- **To Instruction Buffers**

Take output from trace cache if trace cache hit; otherwise, take output from instruction cache.
Multiple Branch Predictor

What Does A Trace Cache Line Store?

- 16 slots for instructions. Instructions are stored in decoded form and occupy approximately five bytes for a typical ISA. Up to three branches can be stored per line. Each instruction is marked with a two-bit tag indicating to which block it belongs.

- Four target addresses. With three basic blocks per segment and the ability to fetch partial segments, there are four possible targets to a segment. The four addresses are explicitly stored allowing immediate generation of the next fetch address, even for cases where only a partial segment matches.

- Path information. This field encodes the number and directions of branches in the segment and includes bits to identify whether a segment ends in a branch and whether that branch is a return from subroutine instruction. In the case of a return instruction, the return address stack provides the next fetch address.

Trace Cache: Advantages/Disadvantages

+ Reduces fetch breaks (assuming branches are biased)
+ No need for decoding (instructions can be stored in decoded form)
+ Can enable dynamic optimizations within a trace
  -- Requires hardware to form traces (more complexity) → called fill unit
  -- Results in duplication of the same basic blocks in the cache
  -- Can require the prediction of multiple branches per cycle
    -- If multiple cached traces have the same start address
    -- What if XYZ and XYT are both likely traces?
Intel Pentium 4 Trace Cache

- A 12K-uop trace cache replaces the L1 I-cache
- Trace cache stores decoded and cracked instructions
  - Micro-operations (uops): returns 6 uops every other cycle
- x86 decoder can be simpler and slower